OpenGL-assisted Occlusion Culling for Large Polygonal Models

Dirk Bartz   Michael Meißner   Tobias Hüttner

WSI/GRIS*
University of Tübingen, Germany

We present an OpenGL-assisted visibility culling algorithm to improve the rendering performance of large polygonal models. Using a combination of hierarchical model-space partitioning, OpenGL-assisted view-frustum culling, and OpenGL-assisted occlusion culling, we achieve a significantly better performance on general polygonal models than previous approaches. In contrast to these approaches, we only exploit common OpenGL features and therefore, our algorithm is also well suited for low-end OpenGL graphics hardware.

Furthermore, we propose a small addition to the OpenGL rendering pipeline to enhance the framebuffer's ability for faster and more detailed occlusion queries.

CCS Categories: I.3.3 [Picture/Image Generation]: Viewing algorithms, Occlusion Culling; I.3.5 [Computational Geometry and Object Modeling]: Object hierarchies; I.3.7 [Three-Dimensional Graphics and Realism]: Hidden Line/Surface Removal;

Keywords: Visibility culling, occlusion culling, hierarchical data structures, OpenGL, sloppy n-ary space partitioning trees.

1 Introduction

Hidden-line-removal and visibility are among the classic topics in computer graphics [13]. A large variety of algorithms are known to solve these visibility problems, including the z-buffer approach [7], the painter algorithm [13], and many more.

Recently, visibility and occlusion culling have been of special interest for walkthroughs of architectural scenes [1, 36, 27] and rendering of large polygonal models [23, 15]. Unfortunately, these approaches are limited to cave-like scenes [23, 38], or require special hardware support [21].

In this paper, we present an algorithm for general visibility queries. This algorithm exploits several OpenGL features in order to obtain faster results for large polygonal models. To show the applicability of our algorithm – in terms of graphics performance – on low-end graphics workstations, we performed all measurements on an SGI O_2/R10000 and an SGI Octane/MXE graphics workstation. Furthermore, we propose an extension to the OpenGL rendering pipeline to add features for improved general occlusion culling.

In a pre-process, the polygonal models are subdivided into sloppy n-ary space-partitioning-trees (snSP-trees). In contrast to most standard partitioning-trees, like the BSP-tree [14], our subdivision is not a precise one; snSP-tree sibling nodes are possibly not disjoint. This is to prevent large numbers of small fractured polygons, which can cause numerical problems and an increase of the rendering load.

During the actual visibility culling, the OpenGL selection buffer is used to implement a view-frustum culling of the nodes of our subdivision tree. Thereafter, the remaining nodes of our snSP-tree are rendered into our implementation of a *virtual occlusion buffer* to determine the non-occluded nodes. Finally, the polygons of the snSP-

nodes considered potentially visible are rendered into the framebuffer.

Overall, our algorithm features:

- **Portability:** Only basic OpenGL-functionality is used for the implementation of the algorithm. No additional hardware support, such as for texture-mapping [40], or special occlusion queries are necessary. Even most low-end OpenGL supporting PC 3D graphics cards are able to use our visibility culling scheme.

- **Adaptability:** Due to the use of the OpenGL rendering pipeline, the presented algorithm adapts easily to any OpenGL graphics card. Features that are not supported in hardware can be disabled, or are realized in software by the OpenGL implementation.

- **Generality:** No assumptions of the scene topology or restriction on the scene polygons are made.

- **Significant Culling:** Although high culling performance is always a trade-off between culling efficiency and speed efficiency, our algorithm obtains high culling performance, while keeping good rendering performance.

- **Well-balanced Culling:** Different computer systems introduce different rendering and CPU performance. The presented algorithm provides an adaptive balancing scheme for culling and rendering load.

Our paper is organized as follows: In Section 2, we briefly outline previous work that has been done in the field of visibility and occlusion culling. Section 3 discusses some scene organization issues and Section 4 presents details of our algorithm. Section 5 analyzes the results and provides a discussion of our algorithm. In Section 6, we discuss hardware related issues of occlusion culling. Specifically, we discuss existing hardware (Section 6.1), and we discuss hardware for advanced occlusion information (Section 6.2). Finally, we state our conclusion and briefly describe future work.

2 Related Work

There are several papers which provide a survey of visibility and occlusion culling algorithms. In [40], Zhang provides a brief recent overview with some comparison. Brechner surveys methods for interactive walkthroughs [6]. Occlusion algorithms for flight simulation are surveyed in [29].

Early approaches are based on culling hierarchical subdivision blocks of scenes to the view-frustum [15]. Although this is a simple but effective scheme for close-ups, this approach is less suited for scenes that are densely occluded, but lie completely within the view-frustum.

In architectural model databases, the scene is usually subdivided into cells, where each cell is associated with a room of the building. For each potential view point of the cells, the potential visible set (PVS) is computed to determine the visibility. Several approaches have been proposed in [1, 36, 27]. However, it appears

---

*Email: {bartz, meissner, thuettne}@gris.uni-tuebingen.de, University of Tübingen, Auf der Morgenstelle 10/C9, D72076 Tübingen, Germany

that the cell subdivision scheme is not suitable for general polygonal scenes without a room-like subdivision, which limits their applicability significantly.

Several algorithms have been proposed in computational geometry. A brief overview can be found in [16]. Coorg and Teller proposed two object space culling algorithms. In [11], a conservative and simplified version of the aspect graph is presented. By establishing visibility changes in the neighborhood of single occluders using hierarchical data structures, the number of events in the aspect graph is significantly reduced.

Secondly, by combining a shadow-frustum-like occlusion test of hierarchical subdivision blocks (i.e., octree blocks), the number of occlusion queries is reduced [12]. However, both algorithms are neither suited for dense occluded scenes with rather small occluders (resulting in a large increase of queries), nor for dynamic scenes.

Cohen-Or et al. proposed an ϵ-Visibility-Culling for distributed client/server walkthroughs [9, 10, 8]. Computing the shadow-frusta for a series of local view points and an occluder permits visibility queries on the local client. While this approach showed good performance with small polygonal scenes ($< 50K$ polygons) in a client/server environment, it seems less suited for large polygonal scenes with more than 500K polygons.

In [24], an occluder database – a subset of the scene database – is selected. During the occlusion culling, the shadow-frusta of the occluders are computed and a scene hierarchy is culled against these shadow-frusta. Overall, the surveyed computational geometry-based visibility approaches only deal with convex occluders, which limits their practical use severely.

In 1993, Greene et al. proposed the hierarchical z-buffer algorithm (HZB) [21, 20, 18], where a simplified version for anti-aliasing is used in [20]. After subdividing the scene into an octree, each of the octants is culled against the view-frustum as proposed in [15]. Thereafter, the silhouettes of the remaining octants are scan-converted into the framebuffer to check if these blocks are occluded. If they are not occluded, their content is assumed to be not occluded as well; if they are occluded, nothing of their content can be visible. The occlusion query itself is performed by checking a z-value-image-pyramid for changes. Unfortunately, this query is not supported by common graphics hardware. Furthermore, maintaining the z-pyramid turns out to be a very expensive operation. However, we consider this algorithm as the inspiring origin of our approach, presented in Section 4.

In [19], Greene presents a hierarchical polygon tiling approach using coverage masks. This algorithm improves the occlusion query of a HZB, due to the two-dimensional character of the tiling. However, the main contribution of this algorithm is an anti-aliasing method, as the algorithm has advantages for very high-resolution images. The strict front-to-back order traversal of the polygons – necessary for the coverage masks – needs some data structure overhead. Building a hierarchy of an octree of BSP-trees limits the application of this algorithm to static scenes. However, some techniques to overcome these drawbacks are discussed by Greene [19].

Naylor presented an algorithm, based on a 3D BSP-tree for the representation of the scene, a 2D BSP-tree as image representation, and an algorithm to project the 3D BSP-tree subdivided scene into the 2D BSP-tree image [31].

Hong et al. proposed a fusion between the hierarchical z-buffer algorithm [21] and the PVS-algorithm in [27]. In this z-buffer-assisted occlusion culling algorithm, a human colon is first subdivided into a tube of cells in a pre-process. Thereafter, the occlusion is determined on-the-fly by checking the connecting portals between these colon cells, exploiting the z-buffer and temporal coherence to obtain high culling performance [23]. Unfortunately, this approach is closely connected to the special tube-like topology of the colon and therefore, is not suited for general occlusion culling problems.

In [38], a voxel-based occlusion culling algorithm is presented. After classifying the scene on a grid of samples of the dataset as void-cells, solid-cells and data-cells, the occlusion is determined in a pre-process for each potential view point. Presumably, this algorithm achieves good results for cave-like scenes, but has a high memory and processing overhead for sparse scenes like the forest scene of Section 5. Therefore, this algorithm is not suited for a general occlusion culling algorithm.

In 1997, occlusion culling using hierarchical occlusion maps (HOM) was presented [40, 39]. Similar to [24], an occluder database is selected from the scene database. Using these occluders, bounding boxes of the potential occludees of the scene database are tested for overlaps, using the image hierarchy of the projected occluders. Strategies for dynamic scenes are presented in [35] and [40]. Sudarsky and Gotsman propose a fast update of the hierarchical data structure, such as the octree block of the HZB. Zhang [40] et al. suggest using each object of a scene as an occluder in the HOM algorithm.

3 Scene Organization

In general, subdivision schemes for general polygonal models are difficult to derive. This results in individual solutions for different datasets. Hong et al. [23] use a technique which subdivides a voxel-based colon dataset along its skeleton. The size of the different subdivision entities depends on how many voxels belong to this entity. In [34], Snyder and Lengyel proposed that the designer of the scene needs to provide the subdivision. Similarly, Zhang et al. used a pre-defined scene database [40]. The most general approach is to subdivide a polygonal model into more or less regular spatial subdivision schemes, such as BSP-trees [14, 31, 19], k-D-trees [5], or Octrees [18, 21]. While these subdivision schemes produce good results on polygonal models extracted by the Marching Cubes algorithm [26] from uniform grid volume datasets – which provide a "natural" subdivision on Marching Cubes cell base, these schemes run into numerous problems on general models. If a polygon of the model lies on a subdivision boundary of an octree block or a BSP-/k-D-tree subdivision plane, it must be split into several smaller triangles, in order to produce the necessary disjoint representation of the bounding volumes. Unfortunately, this procedure can increase the number of small and narrow polygons tremendously.

Several approaches from collision detection propose different techniques to generate hierarchical subdivisions. In [17], Gottschalk et al. use statistical methods to derive a tree hierarchy of oriented bounding boxes (OBBTree). Barequet et al. use the BOX-TREE data structure to provide subdivisions similar to the OBB-Tree and to the octree scheme [2]. While the previous approaches use oriented bounding boxes (OBB) or axis-aligned bounding boxes (AABB), Klosowski et al. use discrete orientation polytopes (k-dops) to generate better fitting convex hull, where the AABB are a special case of the k-dops [25].

3.1 Sloppy N-ary Space Partitioning Trees

In our approach, we propose the use of a sloppy n-ary Space Partitioning tree (snSP-tree). The geometry of the scene is usually organized in the leaf nodes (geometry nodes) of the tree structure. The upper or inner nodes (subdivision nodes) represent the bounding volume of their child nodes in the subtree.

Generally, this tree structure is similar to octrees, BSP-trees, k-D-trees, or other bounding volume representations. In contrast, a node in a snSP-tree does not have a fixed number of child nodes, hence the name **n-ary** Space Partitioning tree. However, the actual difference (to octrees, BSP-trees, or k-D-trees) is given by the *sloppiness* of the partitioning scheme, where the bounding volumes of tree nodes of the same tree level are not necessarily disjoint. Hence,

polygons which extend into the bounding volume of another sub-division node, do not need to be split into two (or more) smaller triangles; instead, the bounding volume of such polygons overlaps into the bounding volume of the other subdivision node, resulting in a non-disjoint partitioning.

Using a snSP-tree as subdivision representation, we can store any give model in such a tree without a re-triangulation of overlapping polygons. Nevertheless, polygons which expand over large parts of a model, i.e. floors, should be subdivided into smaller polygons to ensure a well balanced tree.

3.2 Scene Subdivision

Using a snSP-tree a subdivision data structure, unfortunately does not solve the actual subdivision problem. However, it removes some of the limitations of other subdivision schemes.

Some semi-automatic subdivision is provided by the SGI OpenGL Optimizer package [33]. Unfortunately, it turned out that most generated subdivisions needed to be tuned in order to achieve reasonably good performance, i.e., minimal bounding volumes. One of the problems is that Optimizer tends to put all geometry – which did not fit into the created scene subdivision – in the right-most branch of the respective subtree, resulting in a poor subdivision quality.

While good results can be achieved using scenes where additional information is available (i.e., medical scanner data (octree or BSP subdivision), or pre-subdivided scenes), the subdivision performance for general models remains improvable.

In our approach, we use a subdivision scheme developed for ray-tracing [30], which we adapted to the needs of occlusion culling [28]. Basically, a set of polygons is subdivided in two or more entities, based on a cost function. At each of the subdivision steps, the polygons of each entity are sorted along the three coordinate axes according to their barycenters. Based on the three sorted lists, potential subdivision planes are evaluated using the cost function.

Although the snSP-tree does not rely on AABB, we use this bounding primitive to be represented by the inner nodes of the snSP-tree.

4 OpenGL-assisted Occlusion Culling

The general strategy of our visibility culling algorithms is similar to the hierarchical approach proposed by Garlick [15] and used by the HZB [21] and the HOM [40]. Occluded objects of a subdivided scene are culled in an initial view-frustum step, and a subsequent occlusion culling step. While Garlick used only view-frustum culling of a hierarchically subdivided scene, Greene and Zhang added an occlusion culling step to the general algorithm. As with most occlusion culling algorithms, the latter two algorithm basically differ by how to detect occluded objects in screen-space. Greene used a z-pyramid to determine changes in the z-buffer indicating a not occluded object. In contrast, Zhang used a hierarchical screen projected map of pre-selected occluders.

In this section, we present a novel solution to the occlusion problem. Our algorithm is based on core OpenGL functionality and utilizes the available capabilities of OpenGL to check for occlusion. As mentioned earlier, we assume a sloppy n-ary Space Partitioning tree (snSP-tree) as a hierarchical representation of a scene, which is generated once per scene in a pre-processing step[1]. For each frame, we perform view-frustum and occlusion culling on this subdivision tree. Figure 1 schematically illustrates the pipeline of our culling algorithm. The individual steps are described in detail in the following sections.

Raw Data

(a) snSP–Tree

View–Frustum Culling

(b) Sparse snSP–Tree

Occlusion Culling

(c) Polygons to render

○ Not visible objects △ Polygons
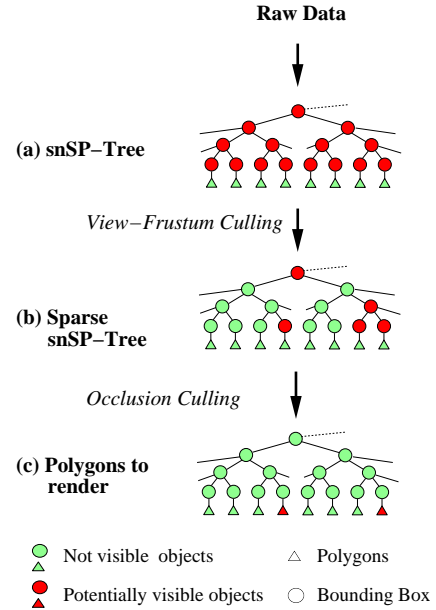● Potentially visible objects ○ Bounding Box

Figure 1: Survey of the basic algorithm.

4.1 View-Frustum Culling

In contrast to other published approaches, we use OpenGL to perform the view-frustum culling step. In detail, we use the *OpenGL selection mode* to detect whether a bounding volume interferes with the view-frustum. This OpenGL mode is designed to identify geometric objects rendered into a specific screen area [37], which in our case is the whole screen. The polygonal representation of the bounding volume (as convex hull) is transformed, clipped against the view-frustum, and finally rendered without contributing to the actual framebuffer [37]. Once a bounding volume intersects the view-frustum – the hit buffer of OpenGL's selection mode has a contribution from this bounding volume object – we test whether the bounding volume resides entirely within the view-frustum. In this case, all subtrees of the bounding volume are marked potentially visible. Otherwise, we recursively continue testing the child nodes of the bounding volume hierarchy.

In rare cases, the bounding volumes can completely contain the view-frustum – resulting in no contributions to the hit buffer of the selection mode, due to a not visible bounding volume representation. This can be prevented by testing if the view point lies within the bounding volume, or if the bounding volume lies in between the near plane of the view-frustum and the view point.

As a result of the view-frustum culling step, leaves are tagged *potentially visible*, if they are not culled by the view-frustum culling, or *definitely not visible*.

4.2 Occlusion Culling

The task of an occlusion culling algorithm is to determine occlusion of objects in a model. We use a *virtual occlusion buffer*, being mapped onto the OpenGL framebuffer to detect possible contribution of any object to the framebuffer. In our implementation of the algorithm on a SGI O_2 and a SGI Octane/MXE, we used the stencil buffer for this purpose[2]. Intentionally, the stencil buffer is used for advanced rendering techniques, like multi-pass rendering.

---

[1] The snSP-tree structure is only generated for static parts of a scene; dynamic parts are handled differently which is discussed in Section 5.1.4.

[2] Other buffers could be used as well, but the stencil buffer, as an integer buffer, is often the least used buffer and has on some graphics systems an

To test occlusion of a node, we send the triangles of its bounding volume to the OpenGL pipeline, use the z-buffer test while scan-converting the triangles, and redirect the output into the virtual occlusion buffer. Occluded bounding volumes will not contribute to the z-buffer and hence, will not cause any trace or *footprint* in the virtual occlusion buffer.

Although reading the virtual occlusion buffer is fairly fast, it is the most costly single operation of our algorithm, which accounts for approximately 90% of the total costs of the occlusion culling stage. This is mainly due to the time consumed for the setup getting the buffer out of the OpenGL pipeline. For models subdivided into thousands of bounding volumes, this can lead to a less efficient operation. Furthermore, large bounding boxes require many read operations. Therefore, we implemented a progressive occlusion test, which reads spans of pixels from the virtual occlusion buffer using a double interleaving scheme, as illustrated in Figure 2.

Although, the setup time on the O_2 for sampling[3] small spans of the virtual occlusion buffer increases the time per sample, spans of ten pixels achieved an almost similar speed-up as sampling entire lines of the virtual occlusion buffer. For the Octane/MXE, we need to read larger chunks from the framebuffer in order to achieve a sufficient speed-up. Overall, there is a trade-off between sampling with good visual quality, and sampling with a minimized setup time.

Span read from the buffer
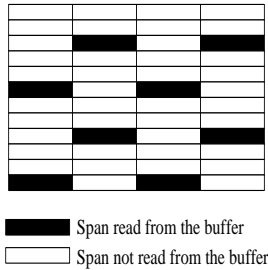Span not read from the buffer

Figure 2: Progressive sampling of the virtual occlusion buffer using a sampling value of 6. Hence, after six sampling iterations, the correct occlusion information will be retrieved.

During motion of the view point, sampling with an appropriate sampling rate enables low culling cost without producing visible artifacts in our scenes. Once the movement stops, the buffer will be read progressively until all values are tested. Basically, every $sampling$th horizontal span is read from the buffer, where the y-offset of this span is incremented by $\frac{sampling}{2}$ for every second column of spans. For the purpose of illustration, Figure 2 uses a sampling factor of six, which is smaller than the sampling factor used in our measurements.

Please note that sampling introduces a non-conservatism into our approach. In some cases, bounding boxes are considered occluded, although they are not fully occluded. However, due to orientation and shape, the actual geometry is usually much smaller than their bounding boxes. Consequently, a not fully occluded bounding box does not mean that the associated geometry is not fully occluded as well. After performing some measurements, it turned out that a sampling factor of ten is sufficient without compromising image quality (see Figure 3). However, the sampling value can be adjusted adaptively.

---

empirically measured better read performance than the other buffers.

[3]Basically, this scheme implements a *sampling* of the virtual occlusion buffer, where $\frac{1}{sampling}$th of each bounding box is read in each iteration. In other words, the algorithm needs *sampling* iterations to fully read the entire bounding box.

4.3 Adaptive Culling

For complex models with deep visibility[4], many almost occluded objects contribute only a few pixels to the final image. Knowing whether an object is not occluded does not introduce a measure of the quantity of contribution. To cull objects which are almost occluded and therefore, are barely noticeable, we introduce adaptive culling as our alternative to approximate culling [40].
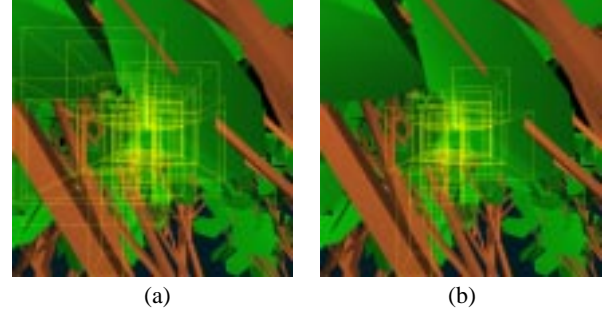

(a)   (b)

Figure 3: Alley of trees – bounding volumes of culled objects are marked yellow: (a) Adaptive culling (94% culled). (b) Occlusion culling (88% culled).

Each bounding volume object of the snSP-tree which generates a footprint on the virtual occlusion buffer needs to be evaluated. Therefore, we count the number of footprints of the object on the virtual occlusion buffer. We consider the depth of the object, the size of its 2D bounding box relative to the view plane, and the number of footprints. In other words, we calculate the percentage of footprints relative to the depth and size of the object.

$$Adap_{cull}(Obj) = \frac{SizeOf2DBoundingBox(Obj)}{SizeOfViewplane} * \frac{Dist(Eye) + Dist(Obj)}{Dist(Eye)} \quad (1)$$

where Sizeof2DBoundingBox(*Obj*) returns the number of pixels of the screen projection of the bounding box, SizeOfViewplane returns the number of pixels of the view plane, Dist(*Eye*) returns the distance between view plane and view point, and Dist(Obj) returns the minimal distance between the *Obj* and the view plane.

For each potentially visible object, we evaluate Equation 1. If $Adap_{cull}(Obj)$ is smaller than a user defined threshold, we consider the object as occluded. This means that an object with a large projected bounding box which is farther away than an object with a similar sized projected bounding box, might have a larger contribution because of its size[5].

Different strategies for dealing with almost occluded objects are possible. First, as mentioned in the previous section, the actual geometry is usually smaller than the associated bounding box. A partially not occluded bounding box does not necessarily mean that the associated geometry is not occluded. Therefore, culling of the object may not have any visual impact. Second, if a small fraction of the actual geometry might be not occluded, we will probably not

---

[4]In scenes with deep visibility, many objects in the background are visible, due to the sparse scene geometry. An example for such a scene is the forest scene in Figure 3 and Figure 8.

[5]Several different heuristics can be used to determine if the contribution of an almost occluded object is significant. In our view, a large but distant object might have a larger visual impact than a small but less distant object (considering the same projected size).

see any detail. Consequently, we could use a lower level of detail representation of this geometry.

Figure 3 shows some results of our adaptive culling mode using the first strategy, compared to the standard occlusion culling mode of our algorithm[6].

4.4 Further Optimizations

Several optimizations are exploited by our approach; many of these ideas are already used by other approaches as well. In this section, we discuss a few of them.

Depth Ordered Culling: Front-to-back, or depth sorted order of the occlusion tests provides a good heuristic for fast filling of the virtual occlusion buffer. Therefore, it is important to process objects in depth sorted order. The z_{min} and z_{max} values for each bounding volume are returned by the view-frustum test for free. The bounding volumes interfering within the view-frustum are sorted by their z_{min} value into a *DepthList*.

Interleaved Culling: Clearly, the subdivision tree representing a model can be too deep to efficiently test every bounding volume for occlusion. In the worst case, each leave could contain a single polygon. This is circumvented by generating well balanced trees, holding sufficient polygons in each leaf. Additionally, the view-frustum culling step and occlusion culling step are dynamically interleaved to exploit culling coherence - an already occluded bounding volume of a tree node does not require any further culling test for its child nodes.

Our occlusion culling step tags each node as potentially visible or occluded. During motion of the view point, those tags have to be updated for every frame. As soon as the camera stops, only bounding volumes, in the previous iteration determined as possibly occluded are progressively refined. Nodes earlier marked potentially visible will stay potentially visible and can therefore be skipped. The leaf nodes which contain the actual geometry are directly sent to the rendering pipeline. This scheme changes once we determine a bounding volume to be potentially visible, which has previously been marked as possibly occluded. In this case, we have to perform occlusion culling for all following nodes in the *DepthList*, due to the changed occlusion in the image. As mentioned earlier, this change of occlusion did not happen in our experiments using a sampling factor of ten.

Cost-adaptive Culling: To obtain a good ratio between time spent for rendering and time spent for culling, we need to ensure that only a reasonable fraction of the rendering time is spent on culling. $F_{graphics}$, the factor which represents the render performance, is hardware dependent and needs to be determined empirically. On the SGI O$_2$, we determined $F_{graphics} = \frac{1}{3}$ as a reasonably good factor.

The cull depth adapts dynamically in order to meet the time budget for culling. This budget is calculated using Equation 2, where T_{render} is the absolute amount of time spent for rendering the previous frame.

$$T_{culling} = T_{render} * F_{graphics} \qquad (2)$$

Once the accumulated culling time of the current frame exceeds $T_{culling}$, the remaining nodes are simply culled against the view-frustum and sent to the rendering pipeline. Furthermore, once a node is detected to be entirely within the view-frustum, all leaves of this node can directly be sent to the rendering pipeline without further view-frustum culling the nodes in between.

---

[6]A threshold of 100 (0.02% of view plane) was used on a view plane of 650 by 650 pixels. Average distance to the objects was 10; their bounding box projection size was on average 423 pixels; the view point was located 0.002 behind the view plane.

Overall Refined Algorithm: To integrate these additional features, the basic algorithm is modified. The *DepthList* is initialized containing at least two uppermost tree nodes which intersect with the view-frustum. Unless the time budget is not entirely consumed, the head element of the *DepthList* is transferred to our cull test. In an interleaved manner, view-frustum culling and occlusion culling for a single frame are performed as described in the following pseudo-code.

```
InitDepthList();
while (UsedTime < Budget)
  Node = DepthList->getHead();
  if (node == LEAF)
    render(Node->polygons);
  else if (OccTest(Node) == NOT_OCCLUDED)
      forall (children(Node))
         if (ViewFrustumTest(child)
               == NOT_OCCLUDED)
          DepthList->add(child);
if (UsedTime < Budget)
  render(remaining_geometry);
```

One advantage of this interleaved culling scheme is the reduced cost for sorting. For a well balanced snSP-Tree of twelve tree levels, we measured for the cathedral scene an average list length of eight and a maximum of 17 potentially visible boundary volumes.

5 Analysis

We examined our algorithm by processing four different scenes; one architectural scene of a 3D array of gothic cathedrals, a city scene, a forest scene to demonstrate adaptive culling, and – similar to [40] – the content of a virtual garbage can of rather small objects.

In this section, we discuss the performance of our algorithm on the test scenes described in Table 1. Note that the achieved percentage of model culled depends on the granularity of the snSP-tree. The more the individual objects of a scene are subdivided, the higher is the potential culling performance. Nevertheless, a higher culling performance does not imply a higher rendering performance. While culling up to 99% of many scenes is possible, the overall rendering performance would drop in most cases. All measurements were performed rendering images of 650×650 pixels on an SGI O$_2$ workstation with 256 MB of memory and an 175 MHz R10000 CPU, and on an SGI Octane/MXE with 896 MB of memory and an 250 MHz R10000 CPU (where noted). Please note that the datasets used for the SGI O$_2$ were using triangle strips, while this was not possible for technical reasons on the Octane.

In the second part of this section, we discuss some limitations of our approach. Finally, we suggest some modifications of the OpenGL rendering pipeline.

scene	#triangles	#objects	#triangles /object
cathedrals	3,334,104	8	416,763
city	1,056,280	300	3521
forest	452,981	12 + 1	28,500 + 110,981
garbage	5,331,146	2,500	about 2,100

Table 1: Model sizes.

5.1 Performance of the Algorithm

5.1.1 Cathedral Scene

In this scene, eight gothic cathedrals are aligned on a $2 \times 2 \times 2$ grid, where each cathedral consists of 416,763 polygons (Figure 13).
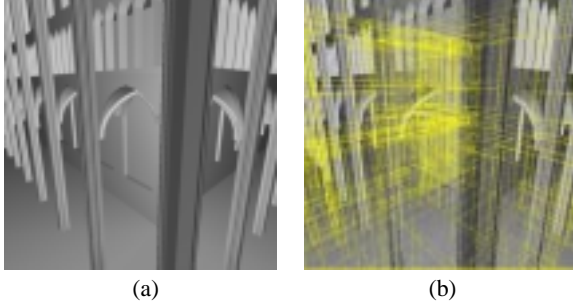
Cathedral Scene

(a) (b)

Figure 4: (a) Interior view of cathedral. (b) Bounding volumes of culled objects are marked in yellow.
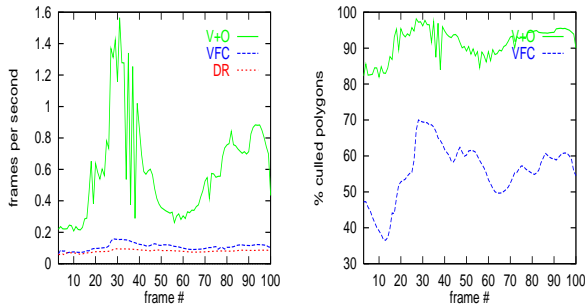

Figure 5: Framerate and percentage of model culled on SGI O_2: V+O denotes view-frustum culling and occlusion culling, VFC denotes view-frustum culling only, and DR denotes direct rendering without any culling.

According to Section 4, we perform two different culling phases, consisting of view-frustum culling and occlusion culling. Figure 5 shows framerate and percentage of model culled of our algorithm on the cathedral scene for a sequence of about 100 frames. For our performance tests, we measured three different modes: **Direct rendering** (DR) – without any culling, **view-frustum culling** only (VFC), and **view-frustum and occlusion culling** (V+O).

The view-frustum only mode culls only small portions of the eight cathedral model; for most view points the other cathedrals are still within the view-frustum. However, occlusion culling is far more successful. Up to an additional 65% of the model is culled away. Due to the occlusion culling, we obtained an average speed-up of seven (Figure 5).
Overall, view-frustum culling accounts for approximately 4.5% of the overall time costs, occlusion culling for 17.1%, and rendering of the potentially visible geometry accounted for approximately 71%.

On the Octane, we achieved an average framerate of 5.3 fps with V+O, resulting in an average speed-up of 12 (see Table 2). The distribution of the time costs for the individual steps was approximately the same as for the O_2.
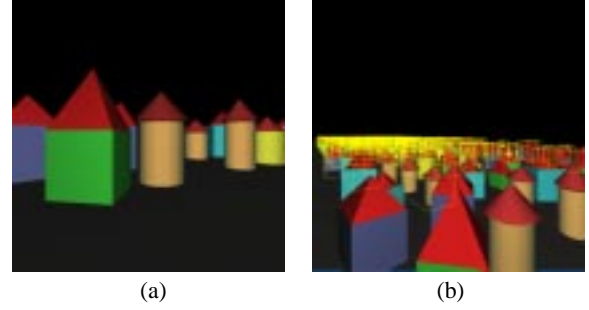
City Model

(a) (b)

Figure 6: City model is rendered using V+O culling: (a) Visitor's view. (b) Bird's-eye view of visitor's view – all yellow bounding volumes are not rendered due to occlusion culling.
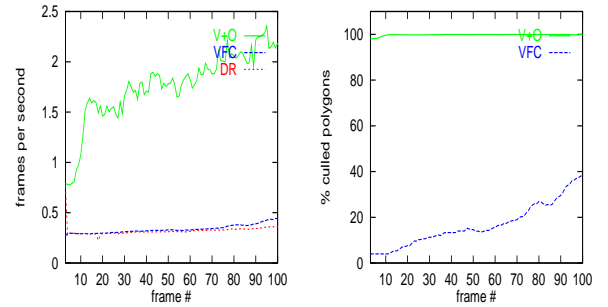

Figure 7: Framerate and percentage of model culled on SGI O_2: V+O denotes view-frustum culling and occlusion culling, VFC denotes view-frustum culling only, and DR denotes direct rendering without any culling.

5.1.2 City Model

The city model is constructed out of three-hundred buildings. Each building contains some interior furniture, which is occluded at all times (Figure 6).

Figure 7 shows framerate and percentage of model culled of the city model. Three culling modes were measured while rendering a sequence of 100 frames: **Direct rendering** (DR) – no culling, **view-frustum culling only** (VFC), and **view-frustum and occlusion culling** (V+O). While the view point is moving near the ground of the scene, 99.8% of the geometry is culled using our culling scheme. Only 3.9% up to 39.9% of the geometry is culled due to view-frustum culling, where the remaining geometry is culled due to our occlusion culling algorithm. On average, we achieved a framerate of almost two frames per second, which represents a speed-up of about eight against view-frustum culling only.
Overall, view-frustum culling accounts for approximately 30.1% of the total rendering time costs, occlusion culling for 61.4%, and the final rendering of the potentially visible geometry only 6.8%.

On the Octane, we achieved an average framerate of 7.7 fps with V+O, resulting in an average speed-up of 9.8 (see Table 2). The distribution of the time costs for the individual steps are 1.4% for view-frustum culling, 7.1% for occlusion culling, and 91.2% for the final rendering.

Forest Scene

(a)   (b)
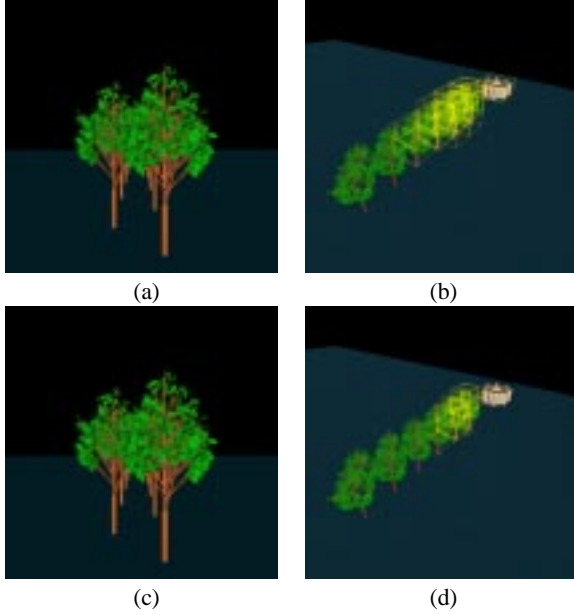
(c)   (d)

Figure 8: The forest scene is rendered using adaptive culling which culled 88% of the structure: (a) Front view. (b) Overview – all culled bounding volumes are marked yellow. (c) The forest scene is rendered using V+O culling which culled 77% of the structure. (d) Overview – all culled bounding volumes are marked yellow.

5.1.3 Forest Scene

The forest scene consists of 12 "tree with leaves" objects – each consists of 28,500 polygons – and one model of "Castle del Monte" of 110,981 polygons behind the trees (Figures 8 and 14). The scattered, yet dense occluded structure of the leaf trees has special demands for an occlusion culling algorithm. Depending on the subdivision of those trees, we achieve higher additional culling, due to adaptive culling; Figure 9 shows an average additional reduction of 11% of the geometry using adaptive culling (AC), compared to the usual V+O culling of our algorithm.

Partially due to the higher framerate of adaptive culling, the overall distribution or time costs is different for adaptive and standard culling. For standard culling, view-frustum culling accounts for approximately 10.1%, occlusion culling for approximately 11.6%, and the final rendering of the potentially visible geometry accounts for approximately 78%. For adaptive culling, view-frustum culling accounts only for 6.4%, occlusion culling for 16.7%, and the final rendering for approximately 76.2% of the total rendering costs.

On the Octane, we achieved an average framerate of 5 fps with AC and 4.7 fps with V+O, resulting in an average speed-up of 6.5 and 6.1 (see Table 2). The distribution of the time costs for the individual steps was approximately the same as for the O_2.

5.1.4 Virtual Garbage Can Scene

To cull dynamic scenes, a special mode can be used. While we use the snSP-tree representation for the static parts of the scene, the dynamic parts are represented by leaf nodes only. The virtual garbage can scene only contains dynamic parts, hence the whole scene is represented by leaf nodes only. We show the performance of this mode on this particular scene in Figure 10. 2,500 independent, potentially moving objects of an average size of about 2,100 polygons are contained in the scene. 96% of the total 5,331,146 polygons are
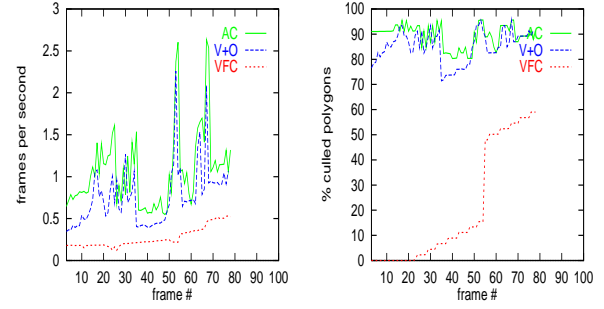

Figure 9: Forest scene; framerate and percentage of model culled on SGI O_2: V+O denotes view-frustum culling and occlusion culling, VFC denotes view-frustum culling only, and AC denotes adaptive culling.

culled. The average obtained speed-up on a SGI O_2 is still larger than seven.

Virtual Garbage Can Scene
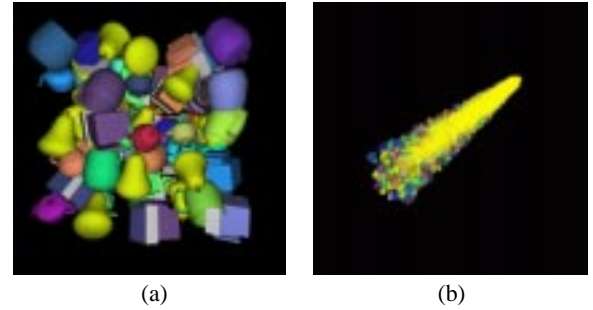

(a)   (b)

Figure 10: (a) Front view. (b) Bird's perspective of front view – all yellow bounding volumes are not rendered due to occlusion culling. Direct rendering took more than 28 seconds, while rendering using our algorithm took less than four seconds.

5.2 Discussion

Most occlusion culling algorithms focus on fast determination if the rendering of bounding volumes changes the content of the framebuffer, hence the associated geometry would be potentially visible [21, 23, 40, 27]. In our approach, we introduced a virtual occlusion buffer which contains the occlusion information. Still, similar to the other approaches, this information has to be read out of the rendering pipeline and searched for changes.

We performed our measurements – in terms of graphics performance – on a low-end graphics workstations, the SGI O_2, and on a SGI Octane/MXE as a mid-range graphics workstation. Similar to PC graphics cards, the O_2 performs parts of the rendering pipeline by the CPU. Only operations associated with the rasterization are executed by special purpose graphics chips, such as framebuffer operation, which leads to fast access to the framebuffer. However, this is not true on highly distributed and interleaved graphics subsystems, like the InfiniteReality graphics of SGI, or to some extent on the Octane/MXE graphics. In these cases, the setup-time for reading the framebuffer are significantly larger, thus limiting the performance of our occlusion culling algorithm. Measurements on the SGI Octane/MXE showed that reading many small portions

scene	#triangles	culling O_2	culling MXE	speed-up O_2/MXE
cathedrals	3,334,104	91.3%	92.5%	4.2 / 12.6
city	1,056,280	99.8%	87.7%	4.8 / 9.8
forest AC	452,981	89.0%	83.0%	3.8 / 6.5
V+C	452,981	84.7%	80.5%	2.6 / 6.1
garbage	5,331,146	96.0%	38.2%	7.0 / 5.0

Table 2: Average performance of OpenGL-assisted Occlusion Culling algorithm compared to view-frustum only culling. The forest scene reflects comparison of adaptive culling (AC) and V+O culling to view-frustum culling.

from the stencil buffer imposes a manifest time penalty. Increasing the size of the samples (and decreasing the number of samples) reduces this overhead, but can lead to severe visual impact, due to large undersampled areas (which are of the same size as the samples). Especially Marching Cubes generated surfaces are sensitive to these undersampled areas. Finding an optimum of a low number of samples while guaranting a sufficient visual quality depends very much on the datasets and the respective graphics hardware.

Overall, our scheme works well on graphics systems based on memory-centered architectures – such as the O_2, the new Visual PC of SGI, or most of the PC graphics cards –, but performs with a reduced speed-up on highly interleaved graphics systems.

6   OpenGL and Occlusion Culling

There are many limiting factors for OpenGL-assisted occlusion culling. Probably most important is the lack of a distinctive hardware supported occlusion culling stage within the rendering pipeline. The strategy presented in this paper so far, as well as others [21, 40], circumvents this by reading values from the framebuffer to detect contributing geometry. Therefore, the ultimate limitation is given by the framebuffer access time. Not surprisingly, read operations are in general relatively expensive and hence, such occlusion culling strategies are not well suited for real-time applications where 30 frames per second and more are required. Dramatic improvements in respect to framerate are usually reported for applications providing hardly any interactivity. Here, the overall framerate can be improved to a peak performance of about five or ten frames per second while not all of the possibly cullable geometry is actually culled. Further improvements could only be achieved if the occlusion query itself is fast, i.e. with real hardware support for occlusion culling.

In the remainder we will discuss available hardware support for occlusion culling and introduce further desirable mechanisms which enable more detailed occlusion queries and hence, additional performance gains.

6.1   Available Hardware Support

In 1997, Hewlett-Packard proposed an extension to OpenGL providing an occlusion query based on a flag [22, 32][7]. Similar to the approach presented in this paper, a bounding volume is rendered without affecting the framebuffer content. This operation is performed in a special OCCLUSION_MODE. While rendering geometry in this mode, a flag will be set in case a fragment passes the depth test of the rendering pipeline. Figure 11 illustrates this process which is located within the per-fragment-stage of the OpenGL pipeline. Basically, the result of the z-buffer comparison is used

---

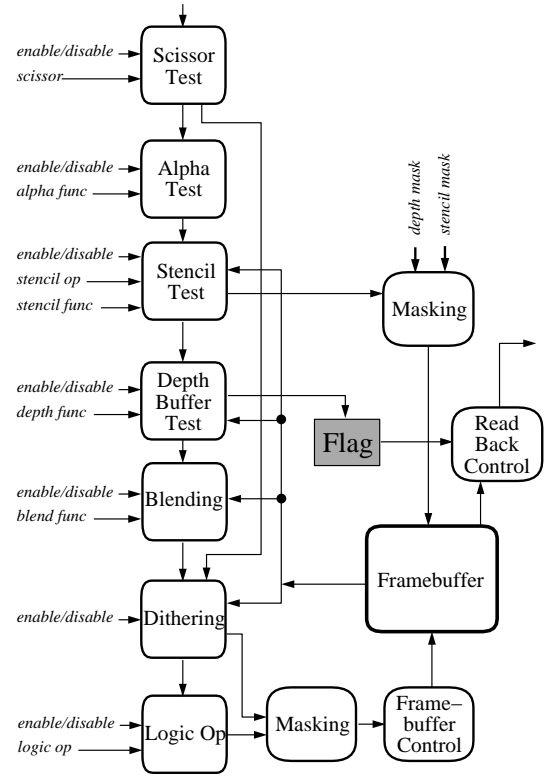[7]SGI's recent Visual PC provides similar functionality.

Figure 11: Schematic of the per-fragment-stage of the graphics pipeline illustrating where the occlusion flag is established.

to establish occlusion information. Thus, the flag is set in the case that any portion of the rendered geometry would contribute to the current content of the framebuffer. Hence, instead of reading entire portions of the framebuffer, the query is reduced to a simple flag test which is very fast.

Depending on the scene, the described occlusion culling method can increase the framerate multiple times of the framerate achieved using view frustum culling only [4]. Here, noticeable framerates of up to 20 Hz can be reported for scenes which are rendered at hardly interactive speed without using this mechanism.

Real-time framerates are somewhat limited since occlusion culling time and actual render time need to be well balanced, thus limiting the amount of occlusion tests which can be performed per frame. Despite the efficient query, establishing the occlusion information is still costly. First, the OCCLUSION_MODE needs to be activated, second all the bounding geometry is scan-converted independent of the state of the occlusion flag, and finally, for non-occluded bounding volumes, valuable render-time is sacrificed without culling any actual geometry. Frequently, bounding volumes are detected visible while being mostly occluded; a visible bounding volume does not necessarily denote that the actual geometry is visible. Tighter bounding volumes might reduce this effect.

Overall, the process of establishing occlusion information is not accelerated since the bounding geometry still needs to be rendered, however, the occlusion query is very fast.

6.2   Advanced Occlusion Information

Instead of using a simple occlusion flag, we propose an entire *Occlusion Unit* which collects more detailed information [3]. In general, the Occlusion Unit should provide information such as number

of visible pixels, closest z-value, number of projected pixels, minimal screen space bounding box, etc. These values can be used to decide in which level-of-detail the possibly visible geometry should be rendered, e.g. if only a few pixels of the bounding volume are visible, the actual geometry could be rendered at a very coarse level of detail. Hence, for visible bounding volumes, rendering of the possibly visible actual geometry can be accelerated by using simplified models, thus achieving higher performance.

To enable the proposed Occlusion Unit, the fragment parameters such as x, y screen space address of the fragment and the corresponding depth value z need to be provided. Additionally, the write enable signal of the depth buffer test, which is used to write and update the framebuffer with the fragment which is closer than the so far stored fragment, needs to be available. Therefore, the Occlusion Unit is located right at the *Depth Buffer Test*, as it is demonstrated in Figure 12. A *projection hit counter* (PHC) counts the number of
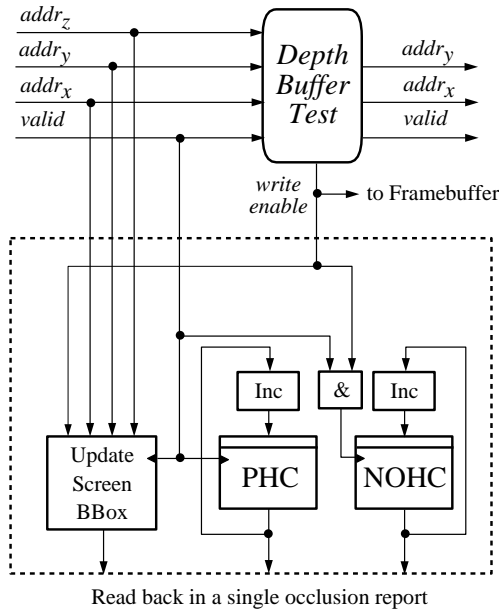

Figure 12: Schematical description of the Occlusion Unit.

projected pixels to give an accurate measure of the projected area in screen space. The *non-occlusion hit counter* (NOHC) counts the number of pixels which actually contribute to the framebuffer. Further registers are needed to store the closest z-value and the minimal screen space bounding box of visible pixels.

For each incoming fragment, the projection hit counter (PHC) is incremented. Furthermore, the non-occlusion hit counter (NOHC) is increased, if the depth buffer test was successful, which denotes the contributing pixels of the framebuffer. To trigger the increment of the non-occlusion hit counter, an AND operation is needed. Besides increasing hit counters, a test is performed whether the screen bounding box defined by the already found non-occlusion hits is increased due to the newly found hit.

The proposed extension can easily be integrated into hardware and will allow for more detailed occlusion queries such that geometry can be rendered at the appropriate level of detail. A very first step towards more detailed occlusion queries is proposed for SGI's visual PC; however only an occlusion flag is supported.

With more detailed occlusion information as described above, non-interactive applications with a high occlusion depth will finally be able to achieve real-time framerates.

7 Conclusion and Future Work

In this paper we have presented a visibility culling algorithm based on core OpenGL functionality. By combining different framebuffer features and sloppy n-ary Space Partitioning-trees – as model-space subdivision – significant culling performance and reasonable framerates were obtained. On average, the culling performance exceeded 90%, while a rendering speed-up factor of almost five – compared to view-frustum culling – was achieved. In comparison, the Hierarchical Occlusion Maps approach [40] achieved speed-ups of 1.5 to 3.5 on a SGI InfiniteReality and MaximumImpact. While our approach encounters some limitations using highly distributed and interleaved graphics hardware (such as SGI's InfiniteReality), it is especially well suited for low- and mid-end graphics subsystems.

Furthermore, we proposed to add features for occlusion culling to the OpenGL rendering pipeline. Specifically, we suggest to add a footprint flag and a footprint counter to qualify and quantify occlusion.

There are still areas for future work on this algorithm. Among the most important are

- **Multi-resolution:** Large model databases usually use multi-resolution methods to represent different levels of detail. Incorporating this level-of-detail functionality is important for the rendering of large-scale scenes, and is of special interest for the adaptive culling mode.

- **Parallelization:** Using a multi-threaded implementation for the occlusion queries promises faster traversal of our model-space snSP-tree. However, using multiple threads for the processing of OpenGL primitives adds a potential bottleneck into our occlusion stage.

- **Scene Organization:** Different scenes of different topology require subdivision schemes of different topology and depth. However, optimizing these structures is not a trivial task and needs further investigation.

Acknowledgements

References

[1] J. Airey, J. Rohlf, and F. Brooks. Towards Image Realism with Interactive Update Rates in Complex Virtual Building Environments. In *Proc. of ACM Symposium on Interactive 3D Graphics*, pages 41–50, 1990.

[2] G. Barequet, B. Chazelle, L. Guibas, J. Mitchell, and A. Tal. BOXTREE: A Hierarchical Representation for Surfaces in 3D. In *Proc. of Eurographics*, 1996.

[3] D. Bartz, M. Meißner, and T. Hüttner. Extending Graphics Hardware for Occlusion Queries in OpenGL. In *Proc. of Eurographics/SIGGRAPH workshop on graphics hardware*, pages 97–104, 1998.

[4] D. Bartz and M. Skalej. VIVENDI - A Virtual Ventricle Endoscopy System for Virtual Medicine. In *Symposium on Visualization*, 1999.

[5] J. Bentley. Multidimensional Binary Search Trees Used for Associative Search. *Communications of the ACM*, 18(9):509–516, 1975.

[6] R. Brechner. Interactive walkthroughs of large geometric databases. In *SIGGRAPH'96 course notes*, 1996.

[7] E. Catmull. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. PhD thesis, University of Utah, 1974.

[8] D. Cohen-Or, G. Fibich, D. Halperin, and E. Zadicario. Conservative Visibility and Strong Occlusion for Viewspace Partitioning of Densely Occluded Scenes. In *Proc. of Eurographics*, pages 243–253, 1998.

[9] D. Cohen-Or and E. Zadicario. On-line Conservative ϵ - Visibility Culling for Client/Server Walkthroughs. In *Late Breaking Hot Topics Proceedings, IEEE Visualization*, pages 37–40, 1997.

[10] D. Cohen-Or and E. Zadicario. Visibility Streaming for Network-based Walkthoughs. In *Proc. of Graphics Interface*, pages 1–7, 1998.

[11] S. Coorg and S. Teller. Temporally Coherent Conservative Visibility. In *Proc. of ACM Symposium on Computational Geometry*, 1996.

[12] S. Coorg and S. Teller. Real-Time Occlusion Culling for Models with Large Occluders. In *Proc. of ACM Symposium on Interactive 3D Graphics*, pages 83–90, 1997.

[13] J. Foley, A. Van Dam, S. Feiner, and J. Hughes. *Computer Graphics: Principles and Practice*. Addison Wesley, Reading, Mass., 2nd edition, 1996.

[14] H. Fuchs, Z. Kedem, and B. Naylor. On Visible Surface Generation by a Priori Tree Structures. In *Proc. of ACM SIGGRAPH*, pages 124–133, 1980.

[15] B. Garlick, D. Baum, and J. Winget. Interactive Viewing of Large Geometric Databases Using Multiprocessor Graphics Workstations. In *SIGGRAPH'90 course notes: Parallel Algorithms and Architectures for 3D Image Generation*, 1990.

[16] J.E. Goodman and J. O'Rourke. *Handbook of Discrete and Computational Geometry*. CRC Press, Boca Raton, 1997.

[17] S. Gottschalk, M. Lin, and D. Manocha. OBBTree: A Hierarchical Structure for Rapid Interference Detection. In *Proc. of ACM SIGGRAPH*, 1996.

[18] N. Greene. *Hierarchical Rendering of Complex Environments*. PhD thesis, Computer and Information Science, University of California, Santa Cruz, 1995.

[19] N. Greene. Hierarchical Polygon Tiling with Coverage Masks. In *Proc. of ACM SIGGRAPH*, pages 65–74, 1996.

[20] N. Greene and M. Kass. Error-bounded Antialiased Rendering of Complex Environments. In *Proc. of ACM SIGGRAPH*, pages 59–66, 1994.

[21] N. Greene, M. Kass, and G. Miller. Hierarchical Z-Buffer Visibility. In *Proc. of ACM SIGGRAPH*, pages 231–238, 1993.

[22] Hewlett-Packard. Occlusion Test, Preliminary. http://www.opengl.org/Developers/Documentation/Version1.2/ HPspecs/occlusion_test.txt, 1997.

[23] L. Hong, S. Muraki, A. Kaufman, D. Bartz, and T. He. Virtual Voyage: Interactive Navigation in the Human Colon. In *Proc. of ACM SIGGRAPH*, pages 27–34, 1997.

[24] T Hudson, D. Manocha, J. Cohen, M. Lin, Kenneth E. Hoff, and H. Zhang. Accelerated Occlusion Culling Using Shadow Frusta. In *Proc. of ACM Symposium on Computational Geometry*, 1997.

[25] J. Klosowski, M. Held, and J. Mitchell. Real-time Collision Detection for Motion Simulation within Complex Environments. In *ACM SIGGRAPH Visual Proc.*, 1996.

[26] W. Lorensen and H. Cline. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. In *Proc. of ACM SIGGRAPH*, pages 163–169, 1987.

[27] D. Luebke and C. Georges. Portals and Mirrors: Simple, Fast Evaluation of Potentially Visible Sets. In *Proc. of ACM Symposium on Interactive 3D Graphics*, 1995.

[28] M. Meißner, D. Bartz, T. Hüttner, G. Müller, and J. Einighammer. Generation of Subdivision Hierarchies for Efficient Occlusion Culling of Large Polygonal Models. Technical Report WSI-99-13, Department of Computer Science, University of Tübingen, 1999.

[29] C. Mueller. Architectures of Image Generators for Flight Simulators. Technical Report TR95-015, Department of Computer Science, University of North Carolina, Chapel Hill, 1995.

[30] G. Müller and D. Fellner. Hybrid Scene Structuring with Application to Ray Tracing. In *Proc. of ICVC*, 1999.

[31] B. Naylor. Partitioning Tree Image Representation and Generation From 3D Geometric Models. In *Proc. of Graphics Interface*, pages 201–212, 1992.

[32] N. Scott, D. Olsen, and E. Gannett. An Overview of the VISUALIZE fx Graphics Accelerator Hardware. *The Hewlett-Packard Journal*, (May):28–34, 1998.

[33] Inc. Silicon Graphics. OpenGL Optimizer Manual. Manual, 1997.

[34] J. Snyder and J. Lengyel. Visibility Sorting and Compositing Without Splitting for Image Layer Decomposition. In *Proc. of ACM SIGGRAPH*, pages 231–242, 1998.

[35] O. Sudarsky and C. Gotsman. Output-Sensitive Visibility Algorithms for Dynamic Scenes with Applications to Virtual Reality. In *Proc. of Eurographics*, pages 249–258, 1996.

[36] S. Teller and C.H. Sequin. Visibility Pre-processing for Interactive Walkthroughs. In *Proc. of ACM SIGGRAPH*, pages 61–69, 1991.

[37] M. Woo, J. Neider, and T. Davis. *OpenGL Programming Guide*. Addison Wesley, Reading, Mass., 2nd edition, 1997.

[38] R. Yagel and W. Ray. Visibility Computations for Efficient Walkthrough of Complex Environments. *PRESENCE*, pages 1–16, 1996.

[39] H. Zhang. *Effective Occlusion Culling for the Interactive Display of Arbitary Models*. PhD thesis, University of North-Carolina at Chapel Hill, 1998.

[40] H. Zhang, D. Manocha, T. Hudson, and Kenneth E. Hoff. Visibility Culling Using Hierarchical Occlusion Maps. In *Proc. of ACM SIGGRAPH*, pages 77–88, 1997.
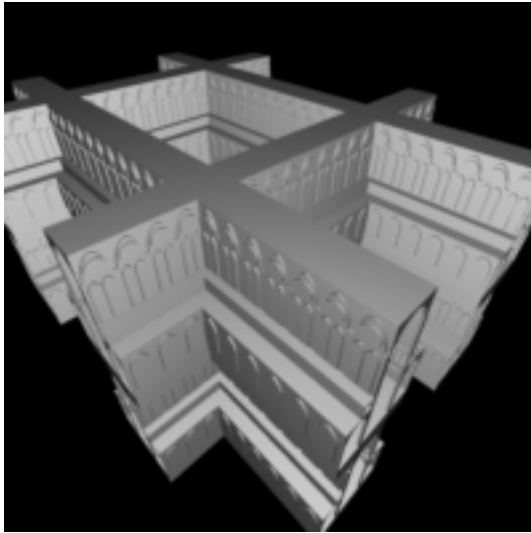
Figure 13: Overview of the cathedral scene.

Figure 14: "Suddenly, the old castle appeared behind the branches of the trees ....."